CS 310 | Spring 2022 | Units 1 & 2 | Lexer & Parser

This document covers two cumulative units. Complete each unit by its respective due date listed on the course calendar.

Follow the instructions using Java source code and PDF documentation. Submit the indicated files (📎) via the assignments in the eCampus lecture section by the due dates on the course calendar. For each submitted file, include your full name as author and a statement acknowledging that your work complies with the academic integrity policy.

If you are pair programming, you and your peer each submit the indicated files independently and are graded independently. For each submitted file, also include your peer's full name as co-author and a description of your and your peer's contributions.

UNIT 1 | LEXER

Topics: *Finite state automata (FSA), lexical analysis, tokens, lexemes*

A [**20%**] Diagram a **finite state automaton** which tokenizes the grammar in this document.
  1 You must complete this diagram individually, because pair programming does not apply.
  2 Illustrate the diagram using professional software and publish it as a 📎 PDF document.
  3 Label each final state with the corresponding token defined in the `model.AbstractLexer.Token` enumeration type.
  4 There is only one diagram. Do not include pseudocode, regular expressions, syntax rules, or semantic rules.

B [**80%**] Implement a **lexer** (lexical analyzer) which tokenizes the grammar in this document.
  1 If you are pair programming, you can complete this implementation with a peer.
  2 Code the class 📎 `unit.Lexer` according to the `model.AbstractLexer` superclass and the Javadoc comments.
  3 Pass all of the `grade.LexerTests` unit tests.
  4 Comply with the code restrictions (see the corresponding section).

C Revise your finite state automaton and your lexer as necessary until they are equivalent to each other.

D Submit the **2 files** indicated above: your FSA and your lexer.

UNIT 2 | PARSER

Topics: *Railroad diagrams (syntax diagrams), LL(1) parsing, recursive descent parsing, precedence, associativity*

E [**20%**] Diagram a set of **railroad diagrams** which recognizes the grammar in this document.
  1 You must complete this diagram individually, because pair programming does not apply.
  2 Illustrate the diagrams using professional software and publish them as a 📎 PDF document.
  3 Each production is a separate diagram, but all diagrams are in one document.

F [**80%**] Implement an LL(1) recursive descent **parser** which recognizes the grammar in this document.
  1 If you are pair programming, you can complete this implementation with a peer.
  2 Code the class 📎 `unit.Parser` according to the `model.AbstractParser` superclass and the Javadoc comments.
  3 Implement left associativity with iteration and right associativity with recursion.
  4 Encapsulate your previously implemented 📎 `unit.Lexer` for lexical analysis (revised as necessary).
  5 Pass all of the `grade.ParserTests` unit tests.
  6 Comply with the code restrictions (see the corresponding section).

G Revise your railroad diagrams and your parser as necessary until they are equivalent to each other.

H Submit the **3 files** indicated above: your railroad diagrams, your parser, and your lexer (whether revised or not).

I During the grade appeal period for UNIT 2, if you earned 70% or more on UNIT 2 but less than 70% on UNIT 1, you can request a **regrade** for UNIT 1. The new grade considers only the revised lexer you submitted for UNIT 2 and counts as a late submission.

CODE RESTRICTIONS

J Do not modify any code in the `grade` or `model` packages.

K Only these types and their API are **permitted** in your code: primitive `int`, `boolean`, and `char` variables, enumeration types, null references, arrays of 1 or 2 dimensions, subclasses of `RuntimeException`, types in the `grade` and `model` packages, and any original types you implement in the `unit` package.

L These types and their API are **forbidden** in your code: boxed `Integer`, `Boolean`, and `Character` objects, strings except as parameters to exception constructors or to tracing framework methods, any unoriginal lexers or parsers such as scanners and regular expressions, and any unoriginal abstract data types such as lists, sets, and maps.

GRAMMAR

This **EBNF grammar with informal semantics** defines a language for *propositional logic with variable assignments*.

- M Terminal symbols are in `blue monospace font` and metasymbols are in `black monospace font`.
- N Whitespace (including spaces, tabs, and new lines) is not a lexeme and can only appear between lexemes, not within.
- O The keywords `let` and `eval` and the operator `v` are case insensitive reserved words.
- P Variable names are case sensitive and must not conflict with any reserved words.
- Q See the unit tests in `grade.LexerTests` and `grade.ParserTests` for **example sentences** in the language.

Syntax			**Semantics**
`<program>`	→	`{ <assignment> }*  <evaluation>`	Initializes a data structure called the lookup table which associates variable names with boolean values for the lifetime of the program, up to a limit of 8 variables Executes each given `<assignment>` in order Returns the boolean result of `<evaluation>`, or throws an `InputException` if the program is syntactically invalid
`<assignment>`	→	`let <variable> = <equivalence> ,`	Associates the new variable name `<variable>` with the boolean value `<equivalence>` in the lookup table, or throws a `VariableException` if the variable name already exists or if the lookup table is already at its limit of variables Returns a void
`<evaluation>`	→	`eval <equivalence> ?`	Returns the boolean result of `<equivalence>`
`<equivalence>`	→	`<implication> { <-> <implication> }*`	Returns the boolean left-associative logical equivalence of `<implication>`$_1$ through `<implication>`$_n$
`<implication>`	→	`<disjunction> { -> <disjunction> }*`	Returns the boolean right-associative logical implication of `<disjunction>`$_1$ through `<disjunction>`$_n$
`<disjunction>`	→	`<conjunction> { v <conjunction> }*`	Returns the boolean left-associative logical disjunction of `<conjunction>`$_1$ through `<conjunction>`$_n$
`<conjunction>`	→	`<negation> { ^ <negation> }*`	Returns the boolean left-associative logical conjunction of `<negation>`$_1$ through `<negation>`$_n$
`<negation>`	→	`<expression> [ ' ]`	Returns the boolean value of `<expression>` or, if the optional terminal is given, its logical negation
`<expression>`	→	`( <equivalence> ) | <boolean>`	Returns the boolean value of the given `<equivalence>` or `<boolean>` respectively
`<boolean>`	→	`1 | 0 | <variable>`	Returns the boolean value `true` if the literal `1` is given or `false` if the literal `0` is given Otherwise, returns the boolean value associated with the existing `<variable>` name in the lookup table, or throws a `VariableException` if the name does not exist
`<variable>`	→	`{ A | B | … | Z | a | b | … | z }+`	Returns the given variable name